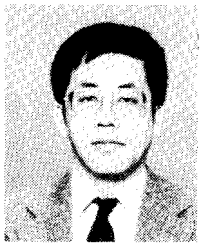


**Fujio Yamamoto** received the B.S. degree in mathematics from Hokkaido University in 1970.

He joined the Central Research Laboratory, Hitachi Ltd., Japan, in 1970. From 1970 to 1976 he was engaged in research and development of a programming language and its application. Since 1977 he has been engaged in the research on large scale circuit simulation system. His current research interests are numerical algorithms for efficient simulation.

**Sakae Takahashi** received the B.E. degree in aeronautics in 1965, the M.E. and the Ph.D. degrees in computer engineering, in 1967 and 1970, respectively, all from Tokyo University.

In 1970, he joined the Central Research Laboratory of Hitachi Ltd., where he is at present engaged in the research and development of supercomputing software. His current research interests include programming languages, compiler design and numerical algorithms for large scale computer simulation of physical phenomena.

# A Hardware Architecture for Switch-Level Simulation

WILLIAM J. DALLY AND RANDAL E. BRYANT, MEMBER, IEEE

*Abstract*—**The Mossim Simulation Engine (MSE) is a hardware accelerator for performing switch-level simulation of MOS VLSI circuits [1], [2]. Functional partitioning of the MOSSIM algorithm and specialized circuitry are used by the MSE to achieve a performance improvement of ≈300 over a VAX 11/780 executing the MOSSIM II program. Several MSE processors can be connected in parallel to achieve additional speedup. A virtual processor mechanism allows the MSE to simulate large circuits with the size of the circuit limited only by the amount of backing store available to hold the circuit description.**

## I. INTRODUCTION

As THE complexity of VLSI circuits approaches $10^6$ devices, the computational requirements of design verification are exceeding the capacity of general purpose computers. To provide the computing power required to verify these complex VLSI chips, special purpose hardware for performing simulation is required. Existing logic simulation engines [3]–[8] are inadequate for MOS VLSI because they cannot accurately model many of the structures found in MOS circuits. Switch-level simulation, on the other hand, more accurately models the effects of capacitance and transistor ratios while operating at speeds comparable to logic simulation.

Existing machines limit size of a circuit which can be simulated by binding circuit element to hardware at compile time. Virtual network processing allows circuits of any size to be simulated by binding circuit elements to hardware at run-time.

Design verification plays an essential role in the development of a VLSI chip. The complexity of the circuits, the inaccessibility of internal nodes, and the difficulty of repair make the probability of producing a working chip very low without extensive design verification. The burden of simulation is compounded by the iterative nature of the design process. Every time an error is discovered in the circuit and the design is modified, all simulations must be repeated to verify that no additional errors have been introduced by the modification. As circuits become more complex, special purpose simulation hardware is required to perform design verification in reasonable time.

A state of the art VLSI chip in 1982 contained $\approx 10^5$ devices and required about 1 week of CPU time to complete a single verification cycle. Since both the number of test vectors required to verify a chip and the amount of computation required to simulate one test vector scale at least linearly with the complexity of a chip, the amount of computation required verify a chip at the switch level scales at least quadratically with complexity. Thus as shown in Fig. 1, a 1986 chip containing $\approx 10^6$ devices will require about 2 years of CPU time to completely verify on a conventional computer. Because of these prohibitive simulation times many groups are abandoning whole chip simulation at the switch level and moving toward mixed-mode simulation. In the long run both mixed-mode simulation and special purpose hardware will be required to meet the growing demands of VLSI.

Conventional simulation engines such as the Yorktown Simulation Engine [3]–[5] and ZyCAD Logic Evaluator

Manuscript received January 20, 1985; revised March 28, 1985. This research was supported by the Defense Advanced Research Projects Agency, ARPA order number 3771, and monitored by the Office of Naval Research under Contract N00014-79-C-0597.

W. J. Dally is with the California Institute of Technology, Pasadena, CA 91125.

R. E. Bryant is with Carnegie-Mellon University, Pittsburgh, PA 15213.

0278-0070/85/0700-0239$01.00 © 1985 IEEE

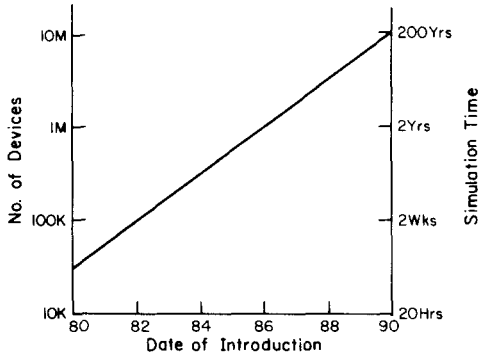


Fig. 1. Scaling of simulation time.

[6] address this problem of long simulation times by performing logic simulation orders of magnitude faster than conventional computers. However, these logic simulation engines do not accurately model the behavior of MOS circuits. Simulation of charge sharing, precharged busses, sneak paths and other subtleties of MOS design are beyond the capabilities of these machines. Switch-level simulation is required to model these effects. Bryant [9]–[12] has developed MOSSIM, a switch-level model and simulator which provides significantly more accurate simulation of MOS circuits than conventional methods of logic simulation [13]. By modeling MOS transistor ratios and node capacitances and by considering a MOS transistor as a truly bidirectional device, MOSSIM provides simulation capabilities which previously required a circuit simulator [14]–[17] with simulation times comparable with logic simulation.

This paper describes the architecture of the MOSSIM Simulation Engine (MSE), a special purpose processor for switch-level simulation of MOS circuits [1], [2]. The MSE combines the speed advantages of hardware simulation engines with the accuracy of the switch-level simulation. The MSE also uses a virtual processor mechanism which allows run-time binding of circuit subnetworks to simulation hardware.

The next section surveys previous work in the field of hardware simulation machines. The conventional simulation engines described suffer both from an inability to accurately model MOS circuits, and from the restrictions of *compile-time* binding of circuit elements and simulation hardware. The MOSSIM algorithm [11] overcomes this first limitation by using a switch-level model of MOS circuits. An adaptation of the MOSSIM algorithm for hardware is described in Section III. Also described in this section is the mechanism of *virtual processors* which allow circuit elements to be assigned to simulation hardware at run-time on a demand basis. Section IV describes the architecture of the MSE. The architecture is developed by observing how potential concurrency in the MOSSIM algorithm can be exploited by hardware. Section V discusses the performance of the MSE. The paper concludes with a status report on the MSE project and some directions for future research.

## II. A Survey of Simulation Machines

Existing logic simulation engines [3]–[8] reduce simulations times by several orders of magnitude compared to conventional computers. The techniques used to achieve this performance improvement can be applied to switch-level simulation as well. In this section we discuss two logic simulation machines: the Yorktown Simulation Engine (YSE) and the ZyCAD Logic Evaluator (ZLE). We compare these machines with the MSE and identify their strengths and weaknesses.

### 2.1. The Yorktown Simulation Engine

The YSE is a multiprocessor composed of up to 256 *logic processors* connected by a crossbar switch [3]–[5]. Each logic processor simulates a subnetwork of up to 4096 four-input gates. The YSE performs either zero delay or unit delay logic simulation using an approach where every gate is simulated every cycle. This *rank order* or *compiled logic* [18] approach is inefficient as the activity in logic circuits exhibits considerable locality and typically no more than 5 to 10 percent of the gates are active at any given time. Although a simulation speed of 12.5 M gate evaluations per second (GEPS) is claimed for the YSE, because of the sparse nature of logic activity typically no more than 1.25M effective GEPS will be achieved.

The YSE is based on a unidirectional gate model which does not lend itself to MOS switch-level simulation. Although a switch-level simulator has been *programmed* on the YSE, by devising logic circuits which model the path strength equations of the switch-level model [11], [19], this approach to switch-level simulation suffers from several limitations. First, since several logic gates are required to simulate a single transistor, the approach is inherently inefficient. Few values of strengths are available, making the modeling of phenomena such as charge sharing difficult. Also, the lack of an event-driven simulation kernel makes it impossible to determine when path finding has completed, forcing all path tracing to be run for a maximum time bound.

### 2.2 The ZyCAD Logic Evaluator

The ZLE is the first commercially available simulation machine [6]. The ZLE consists of up to 15 processors each of which simulates a subnetwork of up to 64 K gates. The machine simulates circuits modeled as a network of three-input gates and one-input bidirectional elements. While technical details of the machine have not been published, the available sales literature [6] indicates that it uses *event driven* [18] simulation which overcomes many of the limitations of the YSE's compiled-logic simulation.

The ZLE includes two features which attempt to bridge the gap between logic simulation and switch-level simulation: bidirectional elements and signal strengths. Later models of the ZLE simulate a bidirectional element. This element propagates logic signals between two I/O ports, $A$ and $B$, with the direction of propagation determined by a control port $C$. This bidirectional element has several shortcomings compared to the switch-level model. Since the bidirectional element only operates one direction at a time, simulation using this model will not detect sneak paths in a circuit. Since the bidirectional element propagates only a logic signal and does not reflect the electrical

TABLE 1
COMPARISON OF SIMULATION ENGINES

| | MSE | YSE | ZLE |
|---|---|---|---|
| Speed/Proc (GEPS) | 250K | 12.5M | 2.5M |
| Gates/Proc | 4K | 4K | 64K |
| Max Gates | none | 1M | 1M |
| Event Driven | yes | no | yes |
| MOS models | yes | no | no |
| Run-Time Binding | yes | no | no |

characteristics of one port to the other, charge sharing effects will not be detected. By not propagating strength information through bidirectional elements, the ZLE cannot model complex pass transistor structures such as barrel shifters and multiplexors with all transistors represented as bidirectional elements. Although these structures do not actually exploit the bidirectional characteristics of MOSFET's, an error in the design might cause a sneak path that would go undetected if it were simulated with unidirectional pass transistor models. Also, since the ZLE does not perform path tracing, an unknown on the control input of a bidirectional element leads to an overly pessimistic propagation of unknowns in the circuit.

The ZLE uses a three-valued strength system where the three strengths correspond to driven, pulled and charged. While this system allows the user to model ratio logic and wire-OR busses, it falls short of the switch-level model in the following areas: Three strengths are usually not sufficient to model the number of different node sizes and transistor strengths in a circuit. The strengths are used only to determine the state of a net and are not propagated through circuit elements.

### 2.3. Comparison

As shown in Table I, the two logic simulation machines described above offer better logic simulation speed than the MSE; however this speed is achieved at the expense of accuracy and flexibility. The MSE is the only machine supporting switch-level MOS models and run-time binding of circuit elements to simulation hardware.

### 2.4. Strengths of Conventional Simulation Engines

Conventional simulation engines are quite successful at improving the performance of logic simulation by orders of magnitude over conventional computers. The techniques used by these machines to achieve speed improvement include: specialization, functional concurrency, subnetwork concurrency and mixed mode simulation. Each of these techniques can be applied to switch-level simulation as well.

*Specialization* involves dedicating special hardware to perform a frequently occurring operation. An example of specialization is the function unit in the YSE which simulates a general four input logic gate in one cycle. Without this special hardware, several cycles would be required to determine the gate output.

*Functional Concurrency* is achieved by performing several of the simulation functions in parallel. An example of functional concurrency is the pipelined gate evaluation in the YSE where several functions (fetch instruction, fetch data, simulate gate, store result) are performed simultaneously. The success of this technique depends on proper balancing of the function units to prevent any one function unit from becoming a bottleneck.

*Subcircuit Concurrency* is achieved by partitioning the circuit into subcircuits and processing each subcircuit simultaneously. Both of the machines described above make some use of subcircuit concurrency. The low activity of logic circuits acts as a two-edged sword in this case. The locality of the logic activity minimizes inter-processor communication preventing processors from idling due to communications delays. However, since only 5 to 10 percent of the logic circuits are active at once, and since the active circuits tend to be clustered, only 5 to 10 percent of the processors may be active at once.

*Mixed Mode Simulation:* The capability to simulate *uninteresting* subcircuits at a high level gives a significant performance improvement. Much less computation is required to simulate one high-level unit (e.g., a 64K RAM) than to simulate many smaller units and their interactions (e.g., the 64K + transistors in the RAM). Mixed-mode simulation also greatly increases the capacity of the machine. Simulating function blocks at a high level leaves more of the capacity of the machine available for the rest of the circuit. The YSE's *array processors* simulate memories at a high level. The logic simulation machine proposed by Abromovici *et al.* [20], [21] incorporates a more complete mixed-mode simulation facility.

### 2.5. Weaknesses of Conventional Simulation Engines

Existing simulation machines suffer from two major weaknesses: 1) they do not accurately model MOS circuits and 2) they bind circuit elements to simulation hardware at compile time.

Logic simulators are inadequate for many MOS circuits which cannot be described in terms of gates. Designers often attempt to force these circuits into a logic model by using several gates to model one transistor [22]. Some logic simulators attempt to provide features such as strengths which mimic the behavior of a switch-level simulator. These approaches in general do not model the subtleties of MOS circuits. Switch-level simulation is fundamentally different from logic simulation. Switch-level simulation operates by tracing paths to solve for the steady-state response of a circuit. This steady-state response is difficult to compute using logic gate models.

All existing simulation machines bind circuit elements to simulation hardware at compile time. In the YSE, even the interconnection performed by the crossbar switch is determined at compile time. This compile time binding limits the size of circuits which can be simulated to the size of the available hardware. Compile time binding also limits the effective concurrency by forcing processors bound to idle subcircuits to remain idle.

The design of the MSE incorporates the speedup techniques used by logic simulation engines and uses the MOSSIM algorithm and run-time binding to overcome their weaknesses. Run time binding attempts to solve the

problem of load balancing. With run-time binding of circuit subnetworks to hardware, a virtual processor mechanism can be implemented where each subnetwork is assigned to a virtual processor, but only the active virtual processors require physical hardware. The use of virtual processors also eliminates an upper bound on the size of a circuit which can be simulated. The virtual processor mechanism is described in more detail in Section III.

## III. Algorithm

The MSE implements the switch-level algorithm described previously [11], referred to here as the "MOSSIM" algorithm. Some modifications were made to the data structures and control flow used in the simulator MOSSIM II, a software implementation of the algorithm, to better explicit the characteristics of hardware. This algorithm is based on a formal switch-level model of MOS transistor networks. By modeling MOS transistor ratios and node capacitances and by considering an MOS transistor as a truly bidirectional device, MOSSIM provides considerably more accurate simulation of MOS LSI than conventional logic simulators. MOSSIM achieves performance comparable with logic gate simulators by using an event-driven scheduling algorithm that exploits the locality of events in a circuit. We will give only an overview of the switch-level model and theory in this paper.

### 3.1. Model

A switch-level network consists of a set of nodes connected by transistors. An *input* node provides a strong, externally set signal much like a voltage source in an electrical circuit. All other nodes are *storage* nodes, able to store a value in the absence of an applied input and to share charge with other storage nodes. Nodal capacitances are modeled by assigning each storage node a *size*, from the set $\{\kappa_1, \cdots, \kappa_{max}\}$ according to the relative capacitance of the node compared to other nodes in the network. Node sizes are ordered, $\kappa_1 < \cdots < \kappa_{max}$. When a set of storage nodes share charge, the resulting state is determined by the state(s) of the largest node(s). Input nodes are indicated by a size $\omega$. Node voltages are represented by states 0 (low), 1 (high), and $X$ (invalid or uninitialized.)

A transistor is a device with terminals labeled, "gate," "source," and "drain" that acts as a resistive switch connecting the source and drain nodes controlled by the state of the gate node. All transistors are viewed as bidirectional elements with no predetermined direction of information or current flow. A transistor has a *type* indicating the conditions under which it will become conducting and a *strength* indicating its conductance relative to other transistors in a ratioed circuit. Transistor types $n$ and $d$ conduct when the gate node is in state 1, while transistor types $p$ and $d$ conduct when the gate node is in state 0. When the gate node of an $n$ or $p$ transistor is in state $X$, the transistor is modeled as having arbitrary conductance between fully conducting and open circuited. Transistor conductances are modeled by assigning each transistor a strength from the set $\{\gamma_1, \cdots, \gamma_{max}\}$ where strengths are ordered $\gamma_1 < \cdots < \gamma_{max}$. In a ratioed circuit consisting of paths of conducting transistors from several input nodes to a storage node, the state of the node is determined by the state(s) of the input node(s) connected by maximum strength paths, where the strength of a path equals the minimum transistor strength in the path. Transistor conduction conditions are represented by states 0 (nonconducting), 1 (fully conducting), and $X$ (between nonconducting and conducting).

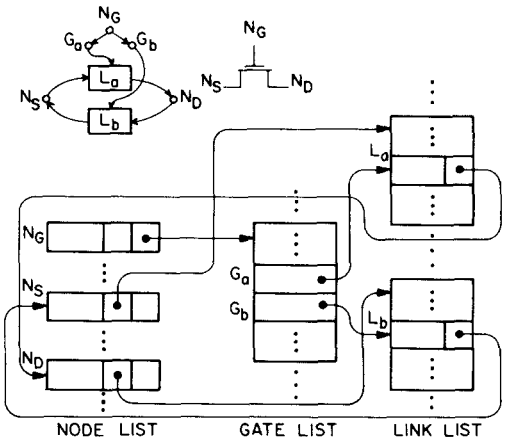



Fig. 2. MSE network data structure.

### 3.2. Data Structure

A switch-level network is represented by three data structures in the MSE, in a manner similar to an adjacency list representation of a directed graph. These data structures are shown in Fig. 2. In fact the simulation algorithm treats the network as a *dynamic multigraph*, where each transistor, depending on its state, can create an edge from its source to its drain and from its drain to its source. A fourth data structure, the stack memory STKM is used to schedule nodes for evaluation. The node list memory, NLM, contains a record for each node, with fields indicating the parameters of the node (size, type, and state), a pointer to the fanout list (representing the set of all transistors for which this node is the gate), and a pointer to the link list (representing the set of all transistors for which this node is the source or drain.) The link list memory, LLM, contains a pair of link records for each transistor: one providing a pointer from the source node to the drain, and one providing a pointer from the drain node to the source. Each link list record also contains fields indicating the transistor parameters (type, strength, and state.) The records in this memory are organized in *link lists*, blocks of contiguous memory locations each containing the set of link records pointing from a given node to all adjacent nodes. The gate list memory GLM contains *gate records*, pointers to the links representing transistors for which a node is the gate. The gate records are organized in *fanout lists*, blocks of contiguous memory locations each containing the set of gate records pointing to all links representing transistors for which a given node is the gate. Thus every transistor in the network requires a two link

records and two gate records in the data structure. By describing the connectivities in the network as sets of pointers organized in blocks, the hardware can traverse the network by following pointers and enumerate connections by sequencing through memory with a counter. By comparison, MOSSIM II also represents the network connectivities as sets of pointers but organizes them in linked lists. We chose a different representation for the MSE to reduce the overhead of the extra memory required to store pointers in a linked list.

### 3.3. Algorithm

To simulate a circuit over a sequence of input vectors, the MSE proceeds as follows:

MSE ALGORITHM-MAJOR LOOP

```
Set all nodes to X
For each input vector do
begin
  Update input nodes
  do begin
    1. Logic Update
    2. Perturbation
    3. Blocking Strength
    4. Up/Down Strength
  end
  until stable state reached or step limit exceeded
end
```

To simulate the response of a circuit to a new set of input values, the MSE repeatedly computes the new excitation states and sets the nodes to their excitation states until a stable state is reached (i.e., the current states equal the excitation states), or a user-specified step limit is exceeded. Computing the excitations involves setting the transistors according to the states of their gate nodes (step 1 in the above program), and computing the *steady-state response* of the nodes (steps 2-4), i.e., the new states formed on the storage nodes due to the connections from input nodes and other storage nodes formed by the conducting transistors.

In general, the algorithm need only compute the steady state response of those nodes that could be affected by the changing conduction states of the transistors and the changing states of the input nodes. All other nodes will have steady state responses equal to their current states. By recomputing the node states only in the active portions of the network, a switch-level simulator can achieve a performance comparable to event-driven logic gate simulators.

Unlike the simulators that propagate logic values through a gate network, MOSSIM computes the steady-state response by analyzing the paths in the graph with edges for each transistor in the 1 or $X$ state between its source and drain nodes. The steady-state responses of the nodes are determined by the paths in the graph as follows. A *path* consists of a root node and a (possibly empty) sequence of edges to a destination node. The strength of a path is defined as the minimum of the size of the root and the strengths of the transistors corresponding to the edges, where strength values are ordered $\kappa_1 < \cdots < \kappa_{max} < \gamma_1 < \cdots < \gamma_{max} < \omega$. This ordering of node sizes relative to transistor strengths reflects the fact that a path of conducting transistors from an input node to a storage node can override any stored charge, and that the state of an input node is not affected by the network connections. A path is *definite* if none of its edges correspond to transistors in the $X$ state. A path is *blocked* if for some initial segment of path (i.e., a path consisting of the same root node and a subset of the edges) and for some definite path with the same destination as the initial segment, the strength of the definite path is greater than that of the initial segment. For a given storage node, its steady state response is determined by the current states of the nodes at the roots of the unblocked paths having this node as destination. That is, the steady state response equals 0 (respectively, 1) if the current states of all the root nodes equal 0 (respectively, 1), and equals $X$ otherwise. It has been shown that this definition in terms of unblocked paths handles ratioed circuits, dynamic charge storage, charge sharing, attenuating pass transistors and unknown logic states in a very accurate manner [11].

The four steps in the above program perform the following functions. During the *Logic Update* step, the conduction states of the transistors whose gate nodes have changed state are updated, and the source and drain nodes of these transistors are queued for the next step. During the *Perturbation* step, the set of all nodes that could be affected by the changing transistor states are found by starting at the nodes queued in the logic update step and traversing the links representing transistors in the 1 or $X$ state. Any path containing an input node other than at the root must be blocked. Hence, the links leading out from an input node are not traversed. Each time a new node is encountered, it is added to the queue. This process continues until no further nodes are found.

The two remaining steps serve to compute the steady-state response for all nodes in the queue. The *Blocking Strength* step determines the strength of the strongest definite path to each node. This computation proceeds in a manner similar to Dijkstra's shortest path algorithm [23], except that rather than finding the path that minimizes the sum of the edge lengths, it finds the path that maximizes the minimum edge strength. That is, for every node $n$, the value block($n$) is initialized to the size of the node and all nodes are placed in a set $S$. The computation proceeds iteratively by selecting and removing a node $n$ from $S$ that maximizes block($n$) and for each link $l$ in the link list for $n$ with state 1 pointing to a node $m$ in $S$, it computes block($m$) $\leftarrow$ max [block($m$), min (block($n$), strength($l$))]. This process continues until the set $S$ is empty. The *Up/Down Strength* step computes the up (respectively, down) value for each node, i.e., the strength of the strongest unblocked path to the node having a root node with state 1 or $X$ (respectively, 0 or $X$). If no such path exists, the value is set to 0, where $0 < \kappa_1$. This computation also proceeds in a manner similar to Dijkstra's algorithm.

For each node $n$, the value up($n$) (respectively, down($n$)) is initialized to the size of $n$ if the current state of node $n$ is 1 or $X$ (respectively, 0 or $X$) and the size of $n$ is greater than or equal to block($n$). Otherwise up($n$) (respectively, down($n$)) is initialized to 0. Starting with the set $S$ containing all nodes, a node $n$ is selected that maximizes the value of max [up($n$), down($n$)] and removed from $S$. For each link $l$ in the link list for $n$ with state 1 or $X$ pointing to a node $m$, it computes a value $v$ = min [up($n$), strength($l$)] and then sets

$$\text{up}(m) \leftarrow \begin{cases} \max [\text{up}(m), v], & v \geq \text{block}(m) \\ 0, & \text{else} \end{cases}$$

and similarly for computing down($m$). If this computation causes the value of up($m$) or down($m$) to change, $m$ must be added back to $S$ if it had previously been removed.[1] Once this computation terminates, the steady state response of node $n$ equals 1 if down($n$) equals 0, 0 if up($n$) equals 0, and $X$, otherwise.

To map the MOSSIM algorithm into hardware, each of the four steps described above was manipulated to fit into a single algorithm template, given below. This factoring of code across the simulation steps allows the same control logic to be used for each simulation step, a criterion more important in hardware design than in software. During the 4 steps, different types of operations are performed, different types of records are followed, and different scheduling criteria are used (indicated by the statements in square brackets), but the overall flow of control is identical.

MSE-ALGORITHM TEMPLATE

```
For each scheduled node n:
  Operate on n
  [schedule n for next step]
  For each active record r in some list for n:
    Operate on dest(r)
    [schedule dest(r) for this step]
    [schedule dest(r) for next step]
```

During the logic update step, no operation is performed on $n$, but $n$ is scheduled for the next step. Then for each record in the fanout list for $n$, the state of the link pointed to is updated. During the perturbation step, the values of block($n$), up($n$), and down($n$) are initialized, and $n$ is scheduled for the next step. Then for each record in the link list for $n$ having state 1 or $X$, the node pointed to is scheduled for the current step. During the blocking strength step, nodes are removed from the queue in decreasing order of their *block* values, and the node is scheduled for the next step. Then for each record in the link list having state 1, the *block* value of the node pointed to is updated as described before. During the up/down strength step, nodes are removed from the queue in decreasing order of the maximum of their *up* and *down* values. For each record in the link list having state 1 or $X$, the *up* and *down* values of the node pointed to are updated as described earlier. If either of these values changes, the node is scheduled for the current step. The new state of the node is computed (even though it may need to be recomputed), and if different from the current state, the node is scheduled for the next step (the up/down step of the next iteration.)

As can be seen by the preceding discussion, the MSE algorithm differs in many respects from conventional logic gate simulators. The fundamental operations involve traversing graphs and computing path strengths. The order in which nodes and links are traversed depends on the dynamic settings of the transistor states, and hence an architecture with a predetermined control flow such as the YSE would not be appropriate. Like logic gate simulation, however, the fundamental operations are quite simple, and the algorithm models all of the nuances of MOS circuits with a single computational paradigm (in terms of unblocked paths). These properties permit a cost-effective hardware implementation that improves greatly on the performance of software switch-level simulators.

### 3.4. *Virtual Processors*

The very locality of activity that enables subcircuit concurrency and localized steady-state response computation can lead to a degradation in performance due to idle processors. If the amount of activity in each processor is not equal, processors with low activity will complete a processing step early and remain idle until all processors complete the step. To avoid this potential degradation, we have developed the concept of *virtual processors* (*VP's*). This concept is analogous to that of *virtual memory* [24]. We partition the circuit into many more subcircuits than we have physical processors. A virtual processor, associated with each subcircuit, contains the complete state of the simulation of that circuit. To maximize throughput, the VP's are dynamically mapped to *physical processors* (*PP's*) based on activity. As soon as a VP becomes idle, it is *swapped out* to be replaced by a VP with activity. This mechanism minimizes the amount of time a PP is idle. This virtual processing method also permits a tradeoff between the amount of hardware (i.e., number of physical processors), and the speed of simulation, without affecting the maximum size circuit that can be simulated. In order to implement the virtual processor concept, a swapping mechanism, mapping mechanism, and scheduling algorithm are required.

*Swapping* is implemented at the VP level by copying the entire state of a VP into backing store. If we restrict swapping to occur at the completion of the outer loop of the algorithm, none of the PP's working registers need be saved or restored. Only the node list, link list, gate list and event stacks (including stack pointers) need be copied. After the old VP is swapped out, the new VP is swapped in by copying its state from backing store.

*Mapping* is required on all interprocessor communica-

[1]These changes to the control flow of Dijkstra's algorithm are required to compute both the up and down values simultaneously.

tions. An individual VP uses only local addresses and requires no mapping. In fact, as long as it is not moved to a different processor group, a VP need not know which PP it is executing on. Mapping is implemented in the interprocessor message switch. All messages are transmitted with virtual addresses. The message switch queues arriving messages according to virtual address. A separate message queue is maintained for each VP. A routing table in the message switch holds the current virtual processor to physical processor mapping and is used to direct output from the queues.

An additional benefit of using short local addresses within VP's and longer global addresses only for interprocessor communication is that the storage requirements for pointers is greatly reduced. Also, the hierarchical addressing mechanism of the MSE is implemented so that it need not be limited to two levels of hierarchy. Messages could be passed between switches using a third level of addressing. Thus, the size of network the MSE can simulate is not limited by address size.

*A Scheduling Algorithm* controls the assignment of virtual processors to physical processors. An efficient scheduling algorithm should optimize throughput by keeping all physical processors busy all the time. A candidate scheduling algorithm is shown below:

VIRTUAL PROCESSOR SCHEDULING ALGORITHM

```
Starting From an Initial Assignment
While some virtual processor is not done
  If a process, p1, on processor P is done
    Select the swapped out process, p2, with the
        longest queue
        Swap p1 out of processor P
        Swap p2 into processor P
```

## IV. ARCHITECTURE

The MSE achieves its performance through subnetwork concurrency, functional concurrency and specialization. Subnetwork concurrency involves partitioning the network into several subnetworks and simulating the subnetworks in parallel. Within each processor functional concurrency is achieved by performing the operations of scheduling, node evaluation and network traversal in parallel. Finally, in performing each of these operations specialized logic circuits are used to implement the time critical functions offering orders of magnitude speedup over general purpose computer instructions.

As shown in Fig. 3, the MSE consists of a number of subnetwork processors (SP) connected by a message bus (MB) to a message switch (MS). Auxiliary processors (AP) may also be connected to the MB to perform functional simulation. A host processor (HP) is connected to all processors by the host bus (HB). The HP controls the operation of the MSE, performs virtual processor swapping and has the ability to read and write each register and memory location in the machine.

The SP's and AP's simulate subnetworks of a circuit in parallel. Interactions between subnetworks create messages which are routed through the MS. The MS performs a virtual subnetwork to physical SP translation for each message and queues messages for subnetworks which are swapped out. Simulation studies indicate that up to eight SP's may be attached to a single MB/MS without significant degradation due to bus contention.

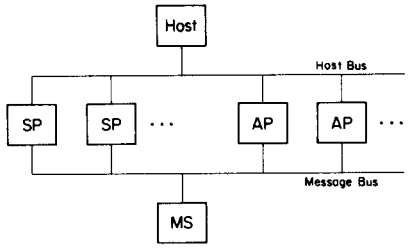



Fig. 3. MSE block diagram.

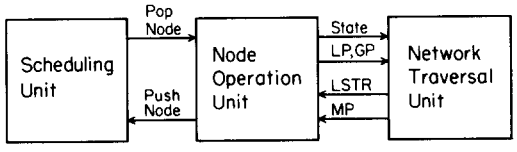



Fig. 4. Subnetwork processor block diagram.

### 4.1. Subnetwork Processor

The SP, shown in Fig. 4, is the hardware kernel of the MSE. Each SP has a capacity of 4096 nodes and 16384 transistors partitioned into separate physical processors of 1024 nodes each. An SP implements the MOSSIM algorithm performing all operations for its subnetwork and sending messages to the MS for operations involving other subnetworks. To exploit all possible functional concurrency a separate function unit is associated with each major data structure. The scheduling unit (SU) implements the scheduling priority queues. The node memory and a path strength unit which operates on nodes are contained in the node operation unit (NOU). The network traversal unit (NTU), contains the link and gate lists which describe the transistors and the network connectivity. The SP also contains two additional function units. The control processor supervises operation of the SP, and the input/output unit handles interprocessor communication. Specialized hardware was added to each unit as necessary to balance their performance so that no one unit was a bottleneck.

#### *Node Operation (NOU)*

The NOU manages the node list data structure performing operating on pairs of nodes as directed by the SU and NTU. The NOU, shown in Fig. 5, consists of a node list memory (NLM), associated addressing registers, and a node data path containing a path strength unit (PSU), and source (SNR), destination (DNR) and result (RRR) registers.

This hardware performs the node operations used to trace paths in the network. It operates by selecting the highest priority node from the active list and computing the effects of this *source node* on all connected destination nodes. If a destination node is updated as the result of this operation, it is added to be active list. First the NOU re-

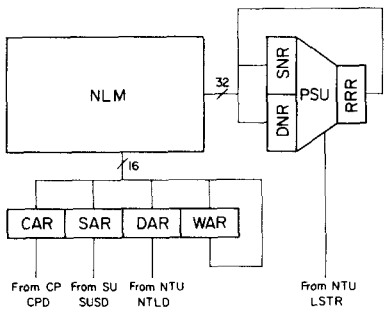



Fig. 5. Node operation unit block diagram.

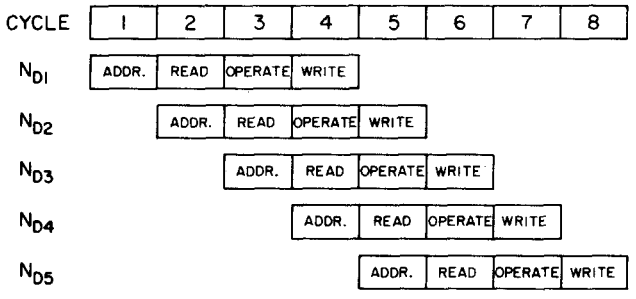



Fig. 6. Destination node operation pipelining.

quests the highest priority node from the SU. The SU responds by returning the address of this *source* node, $N_S$. The address is latched in an addressing register and used to read the source node record from the NLM. This record is latched in the SNR and its link and gate pointers are passed to the NTU. The NTU returns the addresses of nodes which are connected to $N_S$ through an on or unknown transistor. As each of these *destination* node addresses is returned to the NOU, it is latched in an addressing register and used to read a destination node, $N_{Di}$, from the NLM. After the node record for $N_{Di}$ is latched in the DNR, the PSU performs the path strength computation and the result is latched in the RRR and then written back to the DNR, and PSU performs the path strength computation and the result is latched in the RRR and then written back to the NLM. If the destination node is updated, its address is placed on the active list so the change can be propagated to its neighbors.

Operations on destination nodes are pipelined four ways, as shown in Fig. 6. While the address of the fourth destination node $N_{D4}$ arrives from the NTU, the record for $N_{D3}$ is read from the NLM, nodes $N_S$ and $N_{D2}$ are operated on in the PSU, the results from operating on destination node $N_{D1}$ are written back to the NLM. Pipelining allows a path strength operation to be performed once per clock cycle after the pipeline is full. There is some overhead, however, for loading new source nodes. This pipelining is an example of fine-grain functional concurrency (generate address, fetch, operate, write result) nested within the higher level functional concurrency of the SU, NOU, and NTU.

Specialization is also used to accelerate the operation of the NOU. The PSU hardware performs the path strength operation in 100 ns. This operations requires $\approx 30$ lines

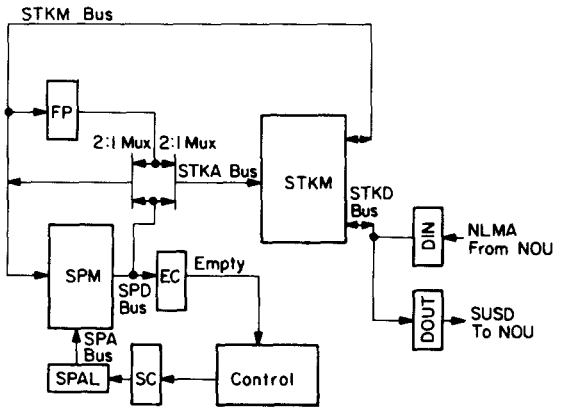



Fig. 7. Scheduling unit block diagram.

of Mainsail [25] code, or $\approx 150$ $\mu$s to perform in software. This speedup, by a factor of 1000, appears to greatly outweigh the contribution of pipelining; however, without pipelining it is not possible to get data to the PSU fast enough to take advantage of its performance. Both functional concurrency and specialization are required to achieve balanced performance.

*Scheduling Unit (SU)*

The SU manages the priority quene data structure used for scheduling operations on nodes. The priority queues are implemented using buckets. Each bucket is a linked list stack allocated from a free list. A separate stack is used for each priority level for both the current and next simulation steps. Since nodes are scheduled according to signal strength, there are sixteen stacks for each step, one stack for each strength. Hooks are provided to add timing simulation by allocating a separate stack for each time increment in the logic update step.

As shown in Fig. 7, the SU consists of a stack memory (STKM), a stack pointer memory (SPM), and a free pointer (FP). To support a linked list structure, each entry in the STKM contains two fields, data and next. The FP points to the beginning of the free list and each SP in the SPM points to to top of a scheduling stack. A strength counter (SC) keeps track of the strongest node scheduled for the current step so that nodes can be retrieved in order of strength. An empty comparator (EC) checks pointers for a null value.

SU operations are initiated by NOU requests to pop the highest priority node form the current step's priority queue or to push a node on either the current or next step's queue. The push operation takes place in one clock cycle. During the read half of the cycle, the FP is used as an address to read the head of the free list from the STKM. At the same time, the strength of the signal being pushed and the current simulation step are used as an address to read the appropriate stack pointer (SP) from the SPM. During the write half of the clock cycle, the links are adjusted ($SP \leftarrow FP$, $FP \leftarrow NEXT$, $NEXT \leftarrow SP$). Then the data and the updated links are written.

The pop operation proceeds in two steps: finding the highest priority nonempty stack, and then removing the

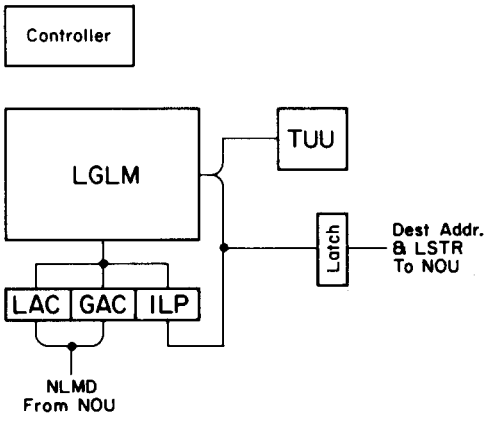



Fig. 8. Network traversal unit block diagram.

head of this stack. The SC, an estimate of the highest priority for which a node is scheduled, is used to address the SPM and is decremented until a nonempty stack is found. Since the SC tracks each push, it is always an upper bound on the highest nonempty priority. Once a nonempty stack is found, the data is read and the pointers are adjusted by reversing the assignments listed above for push.

Specialization is used in the SU to match its speed to the requirements of the NOU. A push takes only one clock cycle, and since the initial value of the SC almost never finds an empty stack the pop operation typically takes two clock cycles. Implementing these operations in software would take about 10 assembly language instructions or 10 $\mu$s.

*Network Traversal Unit (NTU)*

The NTU manages the network topology data structure. It traverses the adjacency list of the current source node returning pointers to destination nodes. During the logic update step the NTU updates the states of transistors controlled by nodes which have changed state. The two operations correspond to the link list and gate list data structures, shown in Fig. 2.

A block diagram of the NTU is shown in Fig. 8. The link and gate lists for each node are stored as arrays in the link and gate list memory (LGLM). The LGLM is addressed by three registers: the gate address counter (GAC), the link address counter (LAC) and the indirect link pointer (ILP). A transistor update unit (TUU) updates transistor states.

In operation, the NOU passes the link and gate pointers of the source node, $N_S$, to the NTU. These pointers are latched into the LAC and GAC. To return an adjacency list the LAC is incremented each cycle to sequence through the source node's link list. Each link in the list is examined to determine if it is active and if it is local or external. The pointers from local active links are returned to the NOU. External active links initiate a message transmission to the destination node's virtual processor.

During the LU step, the NTU sequences through a node's gate list and updates all the destination link records with their new states. First the source node's gate

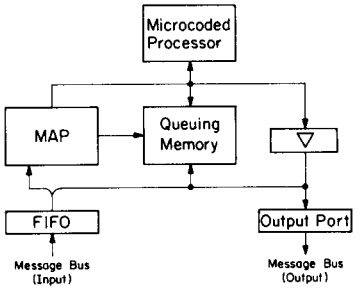



Fig. 9. Message switch, block diagram.

pointer is latched in the GAC. As the GAC sequences through the gate list, the link pointer in each gate record is latched into the ILP. This link is then read from the LGLM, modified in the TUU and written back. If the state of a link changes, a pointer to it destination node is passed to the NOU so it can be scheduled for reevaluation.

Arrays are used to implement the lists in the NTU because these lists are static. Linked lists are used in the SU to allow memory to be dynamically allocated among the lists. Implementing a list as an array allows it to be quickly sequenced by a counter and avoids the overhead of storing next pointers; however changing the size of an array requires copying. While the linked list implementation incurs the overhead of next pointers it is required in the SU where the distribution of memory among the event lists varies considerably with time.

### 4.2. *Message Switch*

The MS routes and queues messages caused by external link and gate records in the NTU. It performs a virtual subnetwork to physical SP translation for each message and queues messages for subnetworks which are swapped out.

As shown in Fig. 9, the message switch consists of an input FIFO, a mapping memory (MM), a queue memory (QM), an output port and a message control processor (MCP). Messages from SP's arrive and are queued in the input FIFO. Each message in turn is then processed by looking up the location of its destination VP in the MM. If the destination VP is resident in a PP, the PP address replaces the VP address in the message, and the message is routed to the output port. If the destination processor is swapped out, the message is queued in the QM. When a processor is swapped in, the MCP transmits its queued messages over the output port.

### 4.3. *Auxiliary Processor*

An auxiliary processor (AP) is planned to provide special purpose hardware to improve the speed of event driven functional simulation. Simulation studies using the DSIM functional simulator [26] have suggested a number of areas where special purpose hardware can accelerate functional simulation. A scheduling unit and network traversal unit similar to the units in the SP would speed up event list management and fanout list traversal. Hardware can speed up message passing between vitual processors and interfacing functional signals to switch-level

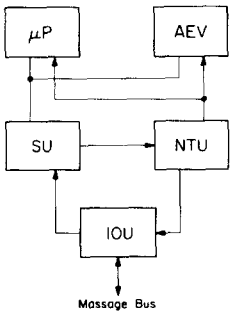



Fig. 10. Auxiliary processor, block diagram.

signals. A signal manipulation unit would accelerate extracting subfields of signals and resolving conflicts on signal nodes. Also models for commonly used functional units such as ROM's, RAM's, and PLA's could be accelerated by hardware.

A block diagram of the AP is shown in Fig. 10. An input/output unit (IOU), handles communication with the message switch. A scheduling unit, (SU), performs event scheduling for logic updates. When an event occurs, the SU sends a signal pointer to the NTU which scans the sensitization list for each signal to determine which functional blocks must be resimulated. Simple functional blocks such as ROMs, RAMs and PLAs will be simulated by an array evaluation unit (AEU). More complex function blocks will be simulated by a 68000 microprocessor.

Adding a functional simulation capability to a simulation accelerator is important for two reasons. First, it allows large circuits to be simulated quickly using mixed mode simulation. More importantly, it allows the input vectors to be generated and output vectors to be checked by functional models within the machine. Without this capability setting input vectors and checking output vectors in the host processor can become a bottleneck.

### *4.4. Host Processor*

The host processor (HP) acts as the user interface to the MSE and the global MSE controller. The host processor performs the following functions:

- command interpretation
- setting nodes for input vectors and watching nodes for output vectors
- loading and partitioning the circuit model
- coordinating SP, MS, and AP operation
- debugging and system diagnostics.

Currently the HP is a SUN-1 workstation running the V-Kernel. We are planning on replacing this host in the near future with a SUN-2 workstation running UNIX.

## V. Performance

The performance of the MSE has been measured at 1.5 M path strength operations per second (PSOPS). This corresponds to about 150 K GEPS for gate oriented circuits, about 300 times faster than a VAX 11/780 running MOSSIM II [12].

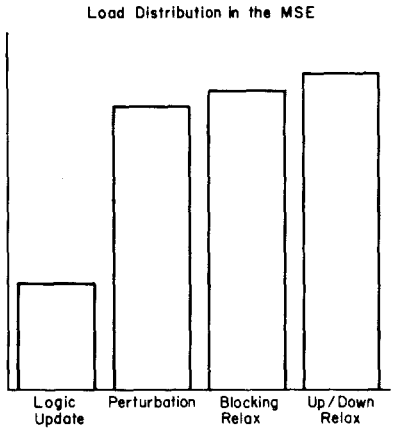



Fig. 11. MSE activity profile.

Compared to the other simulation engines shown in table 1, the MSE appears to be an order of magnitude lower in performance. However, efforts to accurately simulate MOS circuits using conventional logic simulators [19], [22] generally require four to six gates to model a single transistor, and several passes through these gates to model a transistor state change. Furthermore, such models are usually not capable of modeling many MOS circuit phenomena such as charge sharing, and sneak paths.Thus for accurate MOS simulation, the MSE offers better performance than any existing simulation machine.

The performance figure of 1.5 M PSOPS was measured on the prototype MSE hardware using small benchmark circuits. While the MSE can achieve speeds of 5 M PSOPS when its node operation pipeline remains full and no input nodes are encountered, in the benchmark circuits these ideal conditions rarely occurred. In practice each node has an average of three neighbors. As soon as the pipeline is filled for one source node, it must idle for three cycles while the next source node is loaded. Also, typically one quarter to one third of the nodes encountered during simulation are input nodes. When an input node is a destination node, the node operation is performed in the opposite direction requiring an additional cycle.

The relationship of 10 path strength operations per logic event was determined by functional simulation of larger benchmark circuits containing a few thousand transistors. The MSE functional simulator is a 5000 line Mainsail program which simulates the MSE at the register transfer level [25]. Across a wide range of circuits ranging from 20 nodes to 2000 nodes in size, functional simulations show that the MSE requires an average of 10 path strength operations per logic event. A logic event is defined as a logic undate of a transistor gate. This type of event often triggers re-evaluation of an entire logic gate and corresponds well with evaluating a multi-input gate.

A breakdown of where the MSE spends its time is shown in Fig. 11. Only 10 percent of the time is spent in the logic update step while over 60 percent of the total time is spent in the path tracing steps. The design of the MSE has been influenced by these statistics with an emphasis being placed on improving the performance of the path finding

process. In the future, we believe that the most performance improvement will result from optimizations in the perturbation step which will reduce the number of nodes which must be re-evaluated during the path tracing steps.

While we have some simulations of MSE's with several SP's, we have not taken sufficient data to make any conclusions about the efficiency of multiprocessing or the performance of the virtual processor load balancing mechanism.

## IV. History and Status

The MSE project was begun in July 1983 with a study of static and dynamic locality in MOS transistor networks. Based on this study, the architecture was defined in early August. To test the architecture the MSE functional simulator was written during August and was operational by early September. From September to February 1984, an implementation of the MSE SP was designed using standard catalog parts. The design uses 396 integrated circuits packaged on a single wire-wrap board. The board was wired in May and debugging was completed in October 1984.

Much work remains to be done. A great deal of system software must be written before the MSE can be used by designers to simulate their chips. It would be valuable to construct a multiprocessor MSE with additional SP's and a MS to test the idea of virtual network processing. Experiments must be performed to determine message and swapping traffic and their dependency on the ratio of physical SP's to virtual subnetworks. More work is also needed on building processors to accelerate functional and circuit simulation so the MSE can be incorporated in a mixed-mode simulation environment.

## VII. Conclusion

We have designed and constructed the MOSSIM Simulation Engine, a special purpose processor to accelerate switch-level simulation of MOS VLSI circuits. The MSE overcomes two limitations of existing simulation engines: The MSE, by using switch level models, provides greater accuracy when simulating MOS circuits. MOS effects such as charge sharing, sneak paths and dynamic storage are correctly simulated. By using virtual network processing the MSE is able to simulate circuits larger than the size of the simulation machine. We conjecture that virtual network processing makes more efficient use of parallel processors.

For a problem to be a candidate for special purpose hardware, it must be computationally demanding, have a stable and structured algorithm, have potential parallelism (both functional and structural), and have some operations which are poorly matched to the capabilities of a general purpose computer. Speedup is achieved through specialization and concurrency. This speedup must be balanced so that specialized logic does not idle waiting for data.

The implementation of special purpose hardware should not simply follow a software implementation. The cost/performance characteristics of hardware and software are very different. In hardware it is important to fit the problem into a uniform execution mechanism and then to accelerate this uniform mechanism. For the MSE, the MOSSIM algorithm was modified to fit an algorithm template. The hardware of the machine was then designed to optimize execution of the template.

The ideas incorporated in the MSE can be applied to many problems of a similar nature. Many algorithms which appear to be irregular can in fact be fit to a template which can form the basis for special purpose hardware. The concept of virtual network processing, run-time binding of subnetworks to hardware, can be applied to any problem which is characterized by sparse clustered activity. Other problems which are candidates for special purpose hardware and virtual network processing include circuit simulation, geometry compaction, and routing. It is interesting to note that these problems also use a sparse dynamic graph data structure and could make use of the scheduling and network traversal units of the MSE.

One direction of future research is to construct more flexible hardware accelerators. There are far too many demanding applications to construct special purpose hardware for each application. We propose constructing special purpose hardware to accelerate operations on common data structures. By combining these accelerators in different ways designers can share hardware much in the same way that programmers share code. For example, many of the components of the MSE are involved in graph operations such as path finding. These components could be packaged as a graph accelerator and combined with other accelerators to address a wider range of problems.

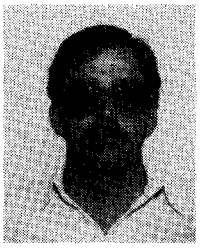
## Acknowledgment

We thank Chuck Seits for providing support and guidance to this project and Hsiu-Tung Yu for contributing to the software and hardware debugging.

## References

[1] W. J. Dally, "The MOSSIM simulation engine: Architecture and design," Caltech Tech. Rep. 5123:TR:84, Apr. 1984.
[2] W. J. Dally and R. Bryant, "A Special purpose processor for switch-level simulation," in *IEEE Int. Conf. on Computer-Aided Design*, pp. 242-244, 1984.
[3] G. F. Pfister, "The Yorktown simulation engine: Introduction," in *19th Design Automation Conf., ACM*, pp. 51-54, 1982.
[4] M. M. Denneau, "The Yorktown Simulation Engine," in *19th Design Automation Conf., ACM*, pp. 55-59, 1982.
[5] E. Kronstadt and G. Pfister, "Software support for the Yorktown simulation engine," in *19th Design Automation Conf., ACM*, pp. 60-64, 1982.
[6] "ZyCad LE-001 and LE-002 product description," ZYCAD, 1982.
[7] R. L. Bartow *et al.*, "Architecture of a hardware simulator," in *IEEE Conf. on Circuits and Computers*, pp. 891-893, 1980.
[8] "Daisy megalogician, product description," Daisy Systems, 1984.
[9] R. Bryant, "A switch-level simulation model for integrated logic circuits," Ph.D. dissertation, MIT, Cambridge, MA, 1981.
[10] ——, "MOSSIM: A switch-level simulator for MOS LSI," in *18th Design Automation Conf., ACM*, pp. 786-790, 1981.
[11] ——, "A switch-level model and simulator for MOS digital systems," *IEEE Trans. Computers*, vol. C-33, pp. 160-177, Feb. 1984.
[12] R. Bryant, M. Schuster, and D. Whiting, MOSSIM II: A switch-level simulator for MOS LSI, User's Manual, Caltech Tech. Rep. 5033:TR:82, Jan. 1983.

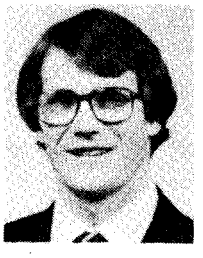
[13] S. A. Szygenda, "TEGAS-2—Anatomy of a general purpose test generation and simulation system for digital logic," in *9th ACM-IEEE Design Automation Workshop*, pp. 116-127, 1972.
[14] A. Vladimirescu *et al.*, SPICE version 2G.5 user's manual, Univ. of California, Berkeley, Tech. Memo., Aug. 1981.
[15] B. Chawla, H. K. Gummel, and P. Kozah, "MOTIS—An MOS timing simulator," *IEEE Trans. Circuits Syst.*, vol. CAS-22, pp. 901-910, Dec. 1975.
[16] A. R. Newton, "Techniques for the simulation of large-scale integrated circuits," *IEEE Trans. Circuits Syst.*, vol. CAS-26, pp. 741-749, Sept. 1979.
[17] A. E. Ruehli and G. S. Ditlow, "Circuit analysis, logic simulation, and design verification for VLSI," *Proc. IEEE*, vol. 71, pp. 34-48, Jan. 1983.
[18] M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*. Computer Sci. Press, 1976.
[19] Z. Barzilai *et al.*, "Simulating pass transistor circuits using logic simulation machines," in *20th Design Automation Conf., ACM*, pp. 157-163, 1983.
[20] M. Abramovici *et al.*, "A logic simulation machine," in *19th Design Automation Conf., ACM*, pp. 65-73, 1982.
[21] M. Abramovici *et al.*, "A logic simulation machine," *IEEE Trans. of Computer-Aided Design*, vol. CAD-2, no. 2, pp. 82-94, April 1983.
[22] W. Sherwood, "A MOS modelling technique for 4-state true-value hierarchical logic simulation," *8th Design Automation Conf., ACM*, pp. 775-785, 1981.
[23] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.
[24] P. Denning, "The working set model for program behavior," *Commun. ACM*, vol. 11, no. 5, pp. 323-333, May 1968.
[25] C. R. Wilcox, M. L. Dageforde, and G. A. Jirak, "Mainsail (TM) language manual version 4.0," *XIDAK*, 1979.
[26] W. J. Dally, "The DSIM functional simulation system: Preliminary user's manual," Caltech Display File, 5109:DF:83, Dec. 1983.



**William J. Dally** received the B.S. degree in electrical engineering from Virginia Polytechnic Institute, Blacksburg, in 1980 and the M.S. degree in electrical engineering from Stanford University in 1981. He is currently working toward the Ph.D. degree in computer science at Caltech.

From 1980 to 1982 he worked at Bell Telephone Laboratories where he contributed to the design of the BELLMAC-32 microprocessor. From 1982 to 1983 he worked as a consultant in the area of digital systems design. His current research interests include computer architecture, computer aided design, VLSI design, and concurrent systems.

*



**Randal E. Bryant** (S'78-M'81) received the B.S. degree in applied mathematics from the University of Michigan, Ann Arbor, in 1973, and the S.M., E.E., and Ph.D. degrees in electrical engineering and Computer Science from the Massachusetts Institute of Technology, Cambridge, in 1977, 1978, and 1981, respectively.

From June 1981 until August 1984 he was an Assistant Professor of Computer Science at the California Institute of Technology, Pasadena. Since September 1984 he has been an Assistant Professor of Computer Science and of Electrical and Computer Engineering at Carnegie-Mellon University, Pittsburgh, PA. His research and teaching interests include VLSI design, verification, and testing, as well as algorithms and computer architecture.

Dr. Bryant is a member of the ACM.

# The *S*-Algorithm: A Promising Solution for Systematic Functional Test Generation

TONYSHENG LIN AND STEPHEN Y. H. SU, SENIOR MEMBER, IEEE

*Abstract*—**We present a new algorithm for functional test generation of VLSI systems. This algorithm systematically finds an input test pattern for each testable register-transfer (RT) level fault defined in our established fault model. The technique developed is appropriate for test generation in top-down Computer-Aided Design process. The development of the algorithm is based on two foundations: the RT-level fault model and symbolic execution technique. A well-defined RT-language for the functional representation of a digital system is described. Based on this language, the RT-level fault modeling and fault collapsing analysis are performed. The fault model is established to lay an analytical foundation for the investigation of faulty behavior among RT-level fault types. The RT-level symbolic execution technique is used to derive test patterns during test generation. Major problem areas are defined and appropriate solutions are presented. The whole test generation process is divided into three stages: preprocess, the *S*-algorithm, and postprocess. "Divide and conquer" principle is used throughout the test generation process for systematic problem solving. The *S*-algorithm is the heart of the overall algorithm. It performs test pattern generation based on the reduced fault model using machine symbolic execution. This test generation algorithm has been implemented in PASCAL on IBM 370/168.**

Manuscript received January 31, 1985; revised March 27, 1985. This paper was written while S. Y. H. Su was a Visiting Professor at the Institut fur Informatik IV, University of Karlsruhe, Germany, under a Alexander von Humboldt Foundation Award for senior U.S. scientists.

T. Lin was with the Computer Science Department, Watson School of Engineering, State University of New York, Binghampton, NY. He is now with AT&T Bell Laboratories, Murray Hill, NJ 07974.

S. Y. H. Su is with the Computer Science Department, Watson School of Engineering, State University of New York, Binghampton, NY 13901.

## I. INTRODUCTION

HARDWARE testing is a generic term which covers multiple activities during the life of a digital system.

0278-0070/85/0700-0250$01.00 © 1985 IEEE